A Spectrum of IV&V Modeling Techniques
NAG-11971

Deliverable

Title: Fault injection tools report

WBS/Task: 4

Date: Sunday, October 05, 2003

Grant

Number: NAG-11971

Project Title: A Spectrum of IV&V Modeling Techniques

Contractor: University of Minnesota

Principal Investigator

Name: Dr. Mats P.E. Heimdahl

Title: Associate Professor

Phone: (612) 625-2068

Fax: (612) 625-0572

Email: heimdahl@cs.umn.edu

Fault Seeding in RSML^{-e} Specifications

Jimin Gao, Robert Weber, and George Devaraj

October 5, 2003

Contents

1 Introduction **1**

2 Background **3**
2.1 Fault Classes 3
2.2 Mutation Analysis 4
2.3 Representative Faults in **RSML^{-e}** 5

3 Proposal **7**
3.1 Mutation Operators 7

4 Prototype Fault Seeder **15**
4.1 Fault Injection Plan 15
4.2 Fault Seeder Design 19
4.2.1 Command Format 19
4.2.2 Mutation Operators 19
4.3 Current Status 33

Chapter 1

Introduction

This RSML^{-e} fault seeding project is intended for generating incorrect specifications and using these specifications to evaluate the realistic performance and fault detection capability of NAYO random search, statistical testing, and various structural coverage criteria. Therefore, our primary goal is to emulate the faults in real RSML^{-e} specifications. At this time, the seeded faults will be specifically used for NAYO search and statistical testing. This report addresses our purpose in seeding faults in specifications, the fault types we are interested in, the mutation operators we propose, open issues in this area to be addressed, and our design approach for fault seeding in RSML^{-e} specifications.

NAYO random search is a technique that first builds the specification model (states and transitions) using a graph notation, and then explores the state space randomly to check for property violations and state reachability. The viability of this approach is based on two observations: firstly, most software faults affect a fair portion of the program state space, and walking through the global state space randomly provides a cost-effective way to search for these faults; Secondly, by randomly walking through the state space, most program local states can be visited in a very short period of time and the number of visited local states reaches a plateau afterwards, which can potentially be exploited by summarizing the property violations as a number of local states. In addition, when the model is large, sometimes it is simply not feasible to build the complete state space using traditional model checking techniques. In some sense, NAYO search performs a Monte Carlo simulation on the model state space to estimate the severity of the faults. However, this approach is not safe due to its incompleteness. Therefore, realistic fault

types and distributions are a necessity for evaluating its performance and fault detection capability.

Fault seeding will also be beneficial for comparing the fault finding ability and performance of different testing strategies that require test case generation. For example, we will compare structural testing against statistical testing. Statistical testing uses an operational profile, the statistical behavior of the system in its environment, to conduct random walks through a system model to generate test cases. This method is expected to be able to generate test cases more quickly than structural testing, but may not reach total coverage of the model without extremely large numbers of test cases. Structural testing can guarantee some level of model coverage but requires more analysis effort to produce the test cases. We will experiment with these strategies to determine their fault detection capabilities and the performance tradeoffs between them. By creating faulty specifications through a well-structured process, we can provide much more realistic and representative testing conditions for comparison.

The generated faults can also be used to test the quality of the verification suites. The set of temporal properties, assumed to express the complete software requirements, may not be able to catch all the faults in reality. An obvious situation is that many of the properties are expressed in terms of RSML^{-e} macros, and a stuck-at-false fault of a macro will not be detected by these properties. Although they are not incorrect technically, they represent a poor integration of model and properties that may create potential loopholes. Therefore, if some changes are not detected by the verification suite, their semantic significance must be manually examined and the overall quality of the verification suite needs to be improved.

Chapter 2

Background

Before deciding on what types of faults to generate, we review some related work on fault classes and mutation analysis, and the observed fault types in real RSML^{-e} specifications.

2.1 Fault Classes

Previous studies have classified software faults based on Boolean formula into four types (classes) [1, 2]:

1. Variable Reference Fault (VRF) - a Boolean variable x is replaced by another variable y, where $x \neq y$.
2. Variable Negation Fault (VNF) - a Boolean variable x is replaced by $\overline{x}$.
3. Expression Negation Fault (ENF) - a Boolean expression p is replaced by $\overline{p}$.
4. Missing Condition Fault (MCF) - a failure to check preconditions.

It has been shown that, for test vector generating purpose, $S_{VRF} \Rightarrow S_{VNF} \Rightarrow S_{ENF}$ and $S_{MCF} \Rightarrow S_{VNF} \Rightarrow S_{ENF}$, where S represents the test conditions for detecting these faults. Namely, a test vector that can detect a variable reference fault or missing condition fault for a Boolean variable occurrence can also detect its variable negation fault and expression negation fault for the expression containing this occurrence. In lieu of performing

a study to determine which fault types subsume which others, we will be seeding faults of various types, trading potentially more test cases and testing time for fault realism.

These studies were intended for deriving test cases from the Boolean formulas to be tested and obtaining a minimal set of test vectors that can detect all possible faults in these formulas, as an extension of traditional hardware testing procedure. Hence, no occurrence statistics for these faults are available. However, these fault types should be considered because Boolean formulas are a major component of any RSML^{-e} specification. In addition, the revealed logical relationship of these faults may imply a relationship of their impacts to the state space of a RSML^{-e} model (the incorrect states introduced by a fault of one type may be a subset of the incorrect states introduced by another fault type on the same Boolean expression), which may in turn significantly affect the performance of NAYO search by creating duplicate faults rather than unique faults.

Similar fault classes were identified in [6] and [8] but in relation to faults found in programs, not specifications.

2.2 Mutation Analysis

Mutation analysis, combined with model checking, has been used to generate test cases for safety property testing [3]. The mutation operators, applied to original expression to generate mutant specifications, were further studied in [4]. The mutation operators used are:

1. Operand Replacement Operator (ORO): replace an operand (a variable or constant) with another syntactically legal operand.

2. Single Expression Negation Operator (SNO): replace a simple expression (a Boolean variable or an expression in the form *token*1 *operator token*2 where *token*1 and *token*2 are variables of scalar type or constants, and *operator* is a relational operator) by its negation.

3. Expression Negation Operator (ENO): replace an expression by its negation.

4. Logical Operator Replacement (LRO): replace a logical operator with another logical operator.

5. Relational Operator Replacement (RRO): replace a relational operator with another relational operator, expect for its opposite.

6. Missing Condition Operator (MCO): delete simple expressions from a Boolean expression.

7. Stuck-At Operator (STO): consist of two operators. Stuck-At-0 replaces a simple expression with 0 and Stuck-At-1 replaces a simple expression with 1.

8. Associative Shift Operator (ASO): change the association between variables.

These mutation operators generally do not correspond exactly to the fault classes discussed in [1]. However, ORO combined with RRO generates a class of faults closely matching VRF. This operator is denoted ORO$^+$.

Empirical studies of these mutation operators on several SMV specification examples showed that ORO$^+$ generated the largest number of inconsistent (*de facto* incorrect) semantically unique mutants and the generated testing traces achieved coverage of 100%, using the metric introduced in [5]. It provided the same set of test cases as all the operators combined. This result is consistent with the theoretical work of [1].

Although these mutation operators correspond to a more general set of fault types than those discussed in section 2.1, they are still mostly focused on the Boolean expressions.

2.3 Representative Faults in RSML^{-e}

The revision histories of ToyFGS02 through ToyFGS05 were examined in order to get a rough idea what types of faults are likely to appear in real RSML^{-e} models. The fault types discovered are:

1. Misspecified conditions. This can be in either a macro or state transition condition.

   (a) Missing conditions: new predicates were later added to an AndOrTable, or a truth value of `*` was changed to `T` or `F`.

   (b) Redundant conditions: conditions were later deleted from an AndOrTable, or a truth value of `T` or `F` was changed to `*`.

(c) Condition negation error: a truth value of `T` was later changed to `F` or a truth value of `F` was changed to `T`.

2. Incorrect initial values. For example, an initial value of a state variable or input variable was changed later from `Undefined` to `True`.

3. Variable-reference error. A misuse of macro or variable names such as `Is_LAPPR_Active` where `Is_LAPPR_Selected` should be used.

Except for the incorrect initial values, most faults appear in AndOrTables, the counterpart of a Boolean condition in other specification languages. The incorrect initial values can be viewed as a special type of operand replacement fault that is unique to RSML^{-e} .

Chapter 3

Proposal

3.1 Mutation Operators

Considering the mutation operators from section 2.2 and the fault types discussed in section 2.3, mutation operators tailored for RSML^{-e} can be defined and applied to generate faulty specifications. Because RSML^{-e} enforces the use of AndOrTables for Boolean conditions by not providing the `AND` and `OR` operators, some of the fault types discussed in section 2.2, such as the associative shift faults and the logic operator replacement faults, cannot appear in a RSML^{-e} specification. They are therefore omitted. Below we show the proposed mutation operators, their justifications and illustrating examples. These operators are defined at the Abstract Syntax Tree (AST) level.

1. Variable Replacement Operator: replace a variable, macro or constant name reference with names of the same type. When applied to AndOrTable predicates, incorrect variable references will be generated. This covers the fault type 3 in section 2.3, one of most frequently seen fault types in RSML^{-e} . For example, the following AndOrTable

```
TABLE
  When_HDG_Activated()    : T * * *;
  When_NAV_Activated()    : * T * *;
  When_LAPPR_Activated()  : * * T *;
  When_LGA_Activated()    : * * * T;
END TABLE
```

can be changed to

```
TABLE
  When_HDG_Activated()    : T * * *;
  When_NAV_Activated()    : * T * *;
  Is_LAPPR_Active()       : * * T *;
  When_LGA_Activated()    : * * * T;
END TABLE
```

2. Condition Insertion Operator: replace a truth value in an AndOrTable, in either a macro or state transition condition, from `*` to `T` or `F`. This will cover fault type 1b in section 2.3. For example, the AndOrTable

```
TABLE
  When_HDG_Activated()    : T * * *;
  When_NAV_Activated()    : * T * *;
  When_LAPPR_Activated()  : * * T *;
  When_LGA_Activated()    : * * * T;
END TABLE
```

   can be changed to

```
TABLE
  When_HDG_Activated()    : T * * *;
  When_NAV_Activated()    : * T * *;
  When_LAPPR_Activated()  : * * T T;
  When_LGA_Activated()    : * * * T;
END TABLE
```

3. Condition Removal Operator: replace a truth value in an AndOrTable, in either a macro or state transition condition, from `T` or `F` or `*`. This will cover fault type 1a in section 2.3. For example, the AndOrTable

```
TABLE
  When_HDG_Activated()    : T * * *;
  When_NAV_Activated()    : * T * *;
  When_LAPPR_Activated()  : * * T *;
  When_LGA_Activated()    : * * * T;
END TABLE
```

can be changed to

```
TABLE
  When_HDG_Activated()    : T * * *;
  When_NAV_Activated()    : * T * *;
  When_LAPPR_Activated()  : * * * *;
  When_LGA_Activated()    : * * * T;
END TABLE
```

4. Condition Negation Operator: replace a truth value in an AndOrTable, in either a macro or state transition condition, from `T` or `F` or from `F` to `T`. This will cover fault type 1c in section 2.3. For example, the AndOrTable

```
TABLE
  When_HDG_Activated()    : T * * *;
  When_NAV_Activated()    : * T * *;
  When_LAPPR_Activated()  : * * T *;
  When_LGA_Activated()    : * * * T;
END TABLE
```

   can be changed to

```
TABLE
  When_HDG_Activated()    : T * * *;
  When_NAV_Activated()    : * T * *;
  When_LAPPR_Activated()  : * * F *;
  When_LGA_Activated()    : * * * T;
END TABLE
```

5. Literal Replacement Operator: a literal value occurrence, in constant definitions, state variable or input variable initial value declarations, state transition target value or source value expressions, or AndOrTable predicates, is replaced with another value of the same type or `Undefined`. This will cover fault type 2 in section 2.3 and some other possible fault types. For example, the following state variable definition

```
STATE_VARIABLE Is_ROLL_Selected: Boolean
   PARENT : NONE
   INITIAL_VALUE : FALSE
   CLASSIFICATION: CONTROLLED

   EQUALS (..ROLL = Selected) IF TRUE
END STATE_VARIABLE
```

can be changed to (two literals are changed for illustrative purpose)

```
STATE_VARIABLE Is_ROLL_Selected: Boolean
   PARENT : NONE
   INITIAL_VALUE : TRUE
   CLASSIFICATION: CONTROLLED

   EQUALS (..ROLL = Cleared) IF TRUE
END STATE_VARIABLE
```

and as another example, the constant definition

```
CONSTANT THIS_SIDE : Side
   VALUE : LEFT
END CONSTANT
```

can be changed to

```
CONSTANT THIS_SIDE : Side
   VALUE : RIGHT
END CONSTANT
```

6. Stuck-at Operator: an AndOrTable, in a macro definition or in a state variable transition condition, is replaced by `True` (stuck-at-true) or `False` (stuck-at-false). Although these types of faults are less likely to appear in real RSML^{-e} specifications, as discussed in section 1, they potentially can be used to evaluate the quality of the test suite. For example, the macro definition

```
MACRO Overspeed_Condition() :
   TABLE
      Overspeed != UNDEFINED  : T;
      Overspeed = TRUE        : T;
   END TABLE
END MACRO
```

can be changed to

```
MACRO Overspeed_Condition() :
   TRUE
END MACRO
```

for a stuck-at-true fault.

7. Predicate Removal Operator: a row in an AndOrTable is removed, resulting in single or multiple missing condition faults. For example, the AndOrTable

```
TABLE
   When_ALT_Switch_Pressed_Seen()              : T *;
   PREV_STEP(Is_VAPPR_Active)                  : F F;
   When_ALTSEL_Target_Altitude_Changed_Seen()  : * T;
   PREV_STEP(Is_ALTSEL_Track)                  : * T;
END TABLE
```

can be changed to

```
TABLE
   When_ALT_Switch_Pressed_Seen()              : T *;
   When_ALTSEL_Target_Altitude_Changed_Seen()  : * T;
   PREV_STEP(Is_ALTSEL_Track)                  : * T;
END TABLE
```

This type of operations are simply synthetic effects of several Truth Value Replacement operations. They are needed however, because intuitively people do make these mistakes when writing specifications.

8. `Not` Operator Removal/Insertion Operator: a `Not` operator is removed from or inserted to a Boolean expression. `NOT` is the only logic operator allowed in RSML^{-e} . It can appear in the AndOrTable predicates, in state variable transition conditions or macro definitions if an AndOrTable is not used, and in state variable transition target value expressions. The removal/insertion of a `NOT` in AndOrTable predicates have the same effect as a Condition Negation operation, so this operator should be used only for full condition expressions and target value expressions. For example, the state variable definition

```
STATE_VARIABLE Independent_Mode: On_Off
   PARENT : None
   INITIAL_VALUE : Off
   CLASSIFICATION: State

   EQUALS On IF Independent_Mode_Condition()
   EQUALS Off IF NOT Independent_Mode_Condition()
END STATE_VARIABLE
```

   can be changed to (both removal and insertion are done for illustrative purpose)

```
STATE_VARIABLE Independent_Mode: On_Off
   PARENT : None
   INITIAL_VALUE : Off
   CLASSIFICATION: State

   EQUALS On IF NOT Independent_Mode_Condition()
   EQUALS Off IF Independent_Mode_Condition()
END STATE_VARIABLE
```

9. Relational Operator Replacement Operator: if its operands are of `Integer` or `Real` type, a relational operator is replaced with another relational operator except for its opposite. This operator generates an incorrect predicate with a mistreated boundary value. For example, the macro definition

```
MACRO AboveThresholdHyst() :
   TABLE
      AltitudeQ1 = Good                            : T F T;
      Altitude1 = UNDEFINED                        : F * F;
      Altitude1 > AltitudeThreshold + Hysteresis   : T * T;
      AltitudeQ2 = Good                            : F T T;
      Altitude2 = UNDEFINED                        : * F F;
      Altitude2 > AltitudeThreshold + Hysteresis   : * T T;
   END TABLE
END MACRO
```

can be changed to

```
MACRO AboveThresholdHyst() :
   TABLE
      AltitudeQ1 = Good                            : T F T;
      Altitude1 = UNDEFINED                        : F * F;
      Altitude1 > AltitudeThreshold + Hysteresis   : T * T;
      AltitudeQ2 = Good                            : F T T;
      Altitude2 = UNDEFINED                        : * F F;
      Altitude2 >= AltitudeThreshold + Hysteresis: * T T;
   END TABLE
END MACRO
```

However, since relational expressions are rare in RSML^{-e} specifications (nonexistent in the FGS models) and numeric values are generally difficult to deal with in model checking, this operator is less important at this stage and its implementation can be postponed.

10. Numeric Operator Replacement Operator: a numeric operator in AndOrTable predicates or in state transition target value expressions can be replaced by another numerical operator. For example, the macro definition shown above can be changed to

```
MACRO AboveThresholdHyst() :
   TABLE
      AltitudeQ1 = Good                            : T F T;
```

```
      Altitude1 = UNDEFINED                       : F * F;
      Altitude1 > AltitudeThreshold + Hysteresis : T * T;
      AltitudeQ2 = Good                           : F T T;
      Altitude2 = UNDEFINED                       : * F F;
      Altitude2 > AltitudeThreshold - Hysteresis : * T T;
   END TABLE
END MACRO
```

For the same reason as discussed above, the implementation of this operator is probably not necessary at this stage.

Chapter 4

Prototype Fault Seeder

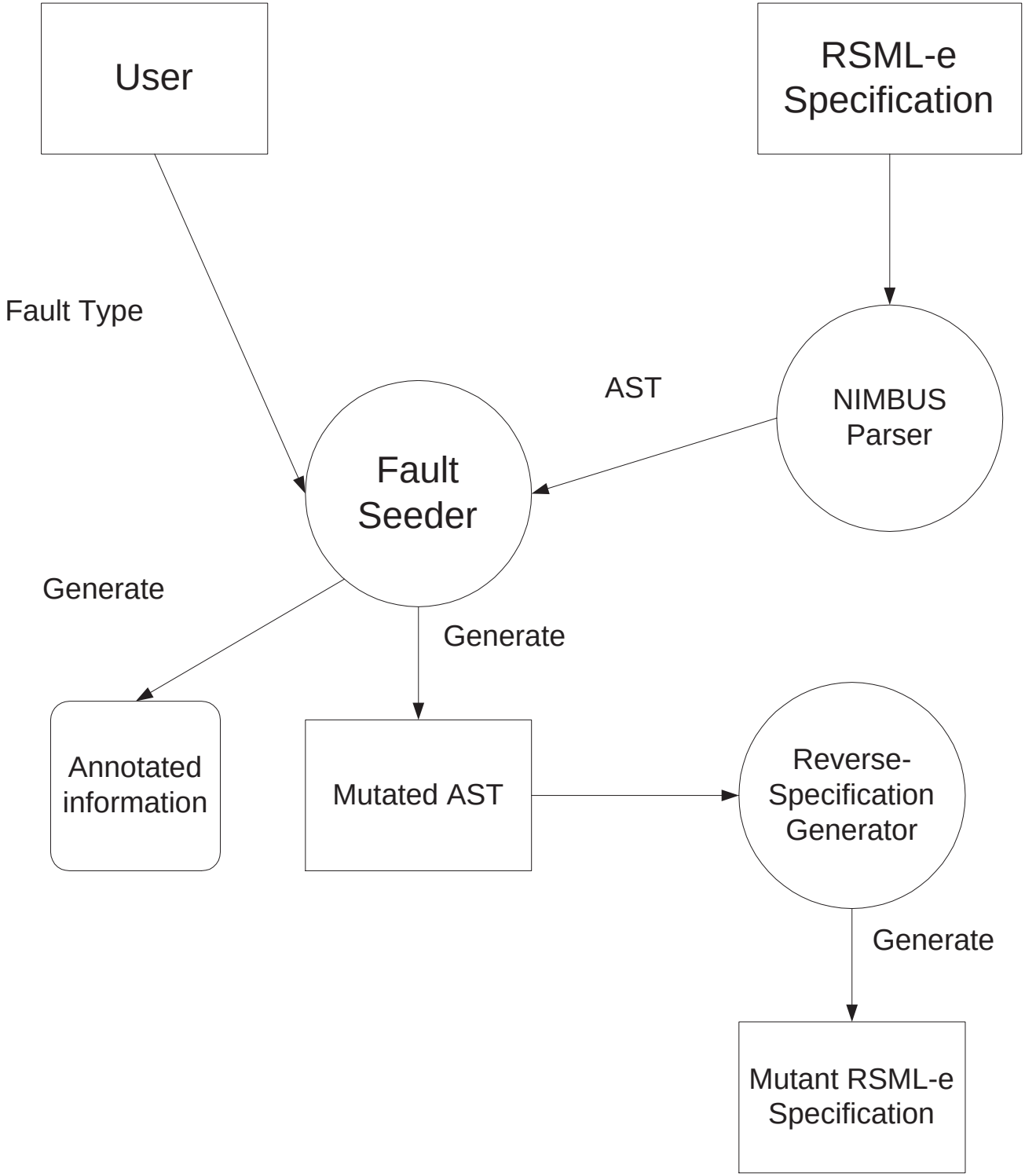
In this section, we discuss the requirements and algorithms for the fault seeder and how we plan to use the fault seeder to generate fault sets that satisfy our needs. The goal of our fault seeder is to produce multiple faulty versions of the specification, each with a single fault of a specified type. The fault seeder can be implemented in NIMBUS using the visitor design pattern and it makes controlled transformations to the internal representation of the specification (Abstract Syntax Tree).The fault seeder has the following components as illustrated in Figure 4.1:

1. The User-Interface component queries the fault type to be seeded and number of faulty versions to be generated for the subject specification.
2. The Fault Seeder performs the fault injection by picking the replacements corresponding to that fault type at random using a random number generator. It also annotates information regarding the location of fault injection and the modification performed.
3. The Reverse-Specification Generator takes the mutated Abstract Syntax Tree and translates it back to the specification in RSML^{-e} , resulting in a mutant RSML^{-e} specification.

4.1 Fault Injection Plan

Step 1: Choose the Artifacts. We intend to apply the fault seeder to a large specification such as the FGS05 specification due to the following

Figure 4.1: Prototype Fault Seeder in NIMBUS


User
RSML-e Specification
Fault Type
AST
NIMBUS Parser
Fault Seeder
Generate
Generate
Annotated information
Mutated AST
Reverse-Specification Generator
Generate
Mutant RSML-e Specification

reasons:

1. Early experiments have shown that for FGS01, SMV without using optimization options has better performance than NAYO. However, for larger-sized specification FGS05, in most test runs NAYO is twice as fast as SMV with options -coi and -dynamic enabled (can be slower sometimes because of the randomness). Since both can process FGS05 in a reasonable amount of time, and to better observe the performance differences, we will not deal with the smaller-sized models.

2. To demonstrate the scalability of the statistical testing approach to larger models and to compare the results obtained with other testing strategies like structural testing.

Step2: Determine number of faults to be seeded corresponding to each fault type. Previous sections identified the faults that are representative of the typically occurring faults. Unfortunately, to date, there is no accepted model with which to determine how many faults of each fault type need to be seeded. Furthermore, the distribution of faults is likely to be significantly different for different developers and different organizations. Therefore, we will seed enough faults of each type of fault to allow for seeding in various levels of the model. Also, the number of faults we seed will be based partly on the number of opportunities for each fault to be seeded. That is, those fault types that have many locations that could be seeded will have many faults seeded; those that have few potential seeding sites will have fewer faults seeded. In reference to our target specification (i.e., FGS05), an exemplary distribution of faults is shown in Table 4.1, which is more or less uniform in nature.

Note that these numbers are just rough figures and were in part determined based on the number of variables, condition tables and predicates in the FGS05 specification. In general, a good estimate as to the number of faults of each fault type to seed can be obtained by a careful examination of the RSML^{-e} specification for the following:

1. Number of variables and variable references.

2. Number of condition tables in the specification.

Fault Type	Number of Faults
Variable Replacements	20
Condition Insertions	20
Condition Removals	20
Condition Negations	20
Literal Value replacements	20
Stuck-at faults	10
Predicate Removals	10
NOT-insertions and deletions	10

Table 4.1: Tentative Fault Distribution for FGS05

3. Number of numeric expressions in the specification. These can occur as predicates in the condition tables or as target expressions in the state transitions.

4. Number of predicates in the condition table.

Alternatively, we can choose a different fault distribution for special purposes. This can be done easily and is something that can be experimented with.

Step3: Perform the randomized fault seeding. The fault transformation component of our fault seeder can be implemented in the NIMBUS environment using a visitor pattern. This visitor would take as input the fault type that the user desires (e.g., macro name replacement), copy the AST, gather the places the fault type could be seeded, and then use random numbers to choose the site to seed and the actual fault that is seeded. The fault transformation component also has the task of annotating the seeded fault and location information in a separate output file.

Step4: Run the mutated Abstract Syntax Tree through the Reverse-Specification Generator. The mutated Abstract Syntax Tree is used by the Reverse-Specification Generator to produce a faulty version or mutant of the original specification.

Fault Type	Command Option
Variable Replacements	-vr
Condition Insertions	-ci
Condition Removals	-cr
Condition Negations	-cn
Literal Value replacements	-lr
Stuck-at faults	-s
Predicate Removals	-pr
NOT-insertions and deletions	-n

Table 4.2: Command Options for Fault Seeding

4.2 Fault Seeder Design

4.2.1 Command Format

The fault seeding command for the NIMBUS toolset will be formulated in the following way:

```
seedfault [option number] ...
```

`option` indicates the type of faults to seed, and `number` specifies the number of faulty specifications to generate for this particular type of fault. Table 4.2 summarizes the command options and the fault types they represent.

4.2.2 Mutation Operators

Each mutation operator proposed in 3.1 can be implemented as a Visitor class. All these Visitors perform similar tasks: they copy the AST, traverse the AST copy to collect the relevant information, create a random mutation of a particular type in the AST copy, and output the mutated AST copy as a corresponding faulty RSML^{-e} specification. This process is repeated as many times as needed to generate the required number of faulty specifications. An explanation of the fault will appear at the location where the change is made as a comment line.

In this document, we only specify the detailed designs of the first six mutation operators, since these would be enough for our current experiments.

Variable Replacement Visitor

To generate a random variable replacement mutation in the AST, the `RVarReplaceVisitor` first traverses the AST. In this process, the pointers to all variable, constant, macro and function reference expressions in AndOrTables are added to a single vector `refs`, and pointers to all variable, constant, macro and function names are grouped into a map `defs` indexed by their types. The `RVarReplaceVisitor` then randomly pick an element from vector `refs` and mutate it with a random element in map `defs` of compatible type. However, we will not attempt to replace a variable reference with macro or functional calls and vice versa, due to the difficulty in replacing an expression node in the AST completely. Instead, we simply mutate the relevant fields of the expression node. In addition, we do not mutate any macro or functional call into a macro or function call with parameters. The C++ style pseudocode for the Variable Replacement Visitor is shown below:

```
#include <vector>
#include <map>
#include <string>
#include <cstdlib>

typedef vector<RDataStructureObject*> nodeVector;
typedef map<string, nodeVector > defMap;

class RVarReplaceVisitor {
private:
    RComponent *fComponent;
    nodeVector refs;
    defMap defs;
    unsigned int faultNumber;

public:
    RVarReplaceVisitor(unsigned int number) {
        fComponent = NULL;
        faultNumber = number;
    }

    virtual ~RVarReplaceVisitor() { }

    void execute(RComponent* component) {
        RCloneComponentPass1Visitor p1_visitor;
        RCloneComponentPass2Visitor p2_visitor;
        RComponent *fComponentCopy;
        RTwoComponentMap* componentMap;
        unsigned int random1, random2, i = 0, flag = 0;

        fComponent = component;

        while (i < faultNumber) {
            // copy the AST
```

```
fComponentCopy = new RComponent();
componentMap = new RTwoComponentMap(fComponentCopy);
p1_visitor.execute(fComponent, fComponentCopy, componentMap);
p2_visitor.execute(fComponent, fComponentCopy, componentMap);

// rename the AST copy
stringstream ss;
ss << fComponent.name() << "_VarReplaceFault" << i;
fComponentCopy->setName(ss.str());

// attempt to visit state variable definitions
{
    StateVariableSet::iterator it;
    for (it = fComponentCopy->stateVariables().begin();
        it != fComponentCopy->stateVariables().end();
        it++) {
            attemptVisit(*it);
        }
}

// attempt to visit input variable definitions
{
    InputVariableSet::iterator it;
    for (it = fComponentCopy->inputVariables().begin();
        it != fComponentCopy->inputVariables().end();
        it++) {
            attemptVisit(*it);
        }
}

// attempt to visit receive type input interfaces
{
    InputReceiverSet::iterator it;
    for (it = fComponentCopy->inputReceivers().begin();
        it != fComponentCopy->inputReceivers().end();
        it++) {
            attemptVisit(*it);
    }
}

// attempt to visit read type input interfaces
...

// attempt to visit output interfaces
...

// attempt to visit macros definitions
...

// attempt to visit function definitions
...

RMacroPred* targetMacroExpr;
RFunctionExpr* targetFuncExpr;
RVariableValueExpr* targetVarExpr;
RConstantExpr* targetConstExpr;
```

```
// select a random element from references
random1 = rand() * refs.size() / (RAND_MAX + 1.0);
targetMacroExpr = dynamic_cast<RMacroPred*>(refs.at(random1));
targetFuncExpr = dynamic_cast<RFunctionExpr*>(refs.at(random1));
targetVarExpr = dynamic_cast<RVariableValueExpr*>(refs.at(random1));
targetConstExpr = dynamic_cast<RConstantExpr*>(refs.at(random1));

// find a random compatible replacement for the reference
if (targetMacroExpr) {
    string s(";;macro");
    defMap::const_iterator iter = defs.find(s);
    if (iter != defs.end()) {
        random2 = rand() * (iter->second.size()) / (RAND_MAX + 1.0);

        if (iter->second.at(random2) != targetMacroExpr->macro()) {
            targetMacroExpr->setMacro(iter->second.at(random2));
            flag = 1;
        }
    }
}
else if (targetFuncExpr) {
    string s = targetFuncExpr->func()->type()->name() + ";;func";
    defMap::const_iterator iter = defs.find(s);
    if (iter != defs.end()) {
        random2 = rand() * (iter->second.size()) / (RAND_MAX + 1.0);

        if (iter->second.at(random2) != targetFuncExpr->function()) {
            targetFuncExpr->setFunction(iter->second.at(random2));
            flag = 1;
        }
    }
}
else if (targetVarExpr) {
    string s = targetVarExpr->variable()->type()->name() + ";;var";
    defMap::const_iterator iter = defs.find(s);
    if (iter != defs.end()) {
        random2 = rand() * (iter->second.size()) / (RAND_MAX + 1.0);

        if (iter->second.at(random2) != targetVarExpr->variable()) {
            targetVarExpr->setVariable(iter->second.at(random2));
            flag = 1;
        }
    }
}
else if (targetConstExpr) {
    string s = targetConstExpr->constant()->type()->name() + ";;var";
    defMap::const_iterator iter = defs.find(s);
    if (iter != defs.end()) {
        random2 = rand() * (iter->second.size()) / (RAND_MAX + 1.0);

        if (iter->second.at(random2) != targetConstExpr->variable()) {
            targetConstExpr->setConstant(iter->second.at(random2));
            flag = 1;
        }
    }
}
```

```
        if (flag) {
            // typecheck the mutated AST
            RParseMessageList l;
            RPP_TypeCheck typeChecker(*newComponent, l);
            typeChecker.execute();

            if (l.errorCount == 0) {
                // print AST to RSML spec
                RPrintASTVisitor pv;
                pv.execute(fComponentCopy);
                i ++;
                flag = 0;
            }
        }

        delete fComponentCopy;
        delete componentMap;
    }
}

void visit(RStateVariable& sv, RVisitParam*) {
    string key = sv.type()->name() + ";;var";
    defMap::iterator iter = defs.find(s);

    // add variable to definition map
    if (iter != defs.end()) {
        iter->second.push_back(&sv);
    }
    else {
        nodeVector n_vec;
        n_vec.push_back(&sv);
        pair<string, nodeVector > p1(key, n_vec);
        defs.insert(p1);
    }

    // iterate through the cases and attempt to visit
    // the AndOrTables in the conditions
    ...
}

void visit(RAndOrTable& table, RVisitParam*) {
    // iterate and attempt to visit the predicates
}

void visit(RVariableValueExpr& expr, RVisitParam*) {
    refs.push_back(&expr);
}

void visit(RMacro& m, RVisitParam*) {
    // add macro to definition map
    string key(";;macro");
    defMap::iterator iter = defs.find(s);

    if (iter != defs.end()) {
        iter->second.push_back(&m);
    }
    else {
```

```
            nodeVector n_vec;
            n_vec.push_back(&m);
            pair<string, nodeVector > p1(key, n_vec);
            defs.insert(p1);
        }

        // attempt to visit the AndOrTable
        ...
    }

    void visit(RCaseFunction& func, RVisitParam*) {
        string key = func.type()->name() + ";;func";
        defMap::iterator iter = defs.find(s);

        // add function to definition map
        if (iter != defs.end()) {
            iter->second.push_back(&func);
        }
        else {
            nodeVector n_vec;
            n_vec.push_back(&func);
            pair<string, nodeVector > p1(key, n_vec);
            defs.insert(p1);
        }

        // attempt to visit the conditions
        ...
    }

    void visit(RMacroPred& expr, RVisitParam*) {
        refs.push_back(&expr);
    }

    void visit(RFunctionExpr& expr, RVisitParam*) {
        refs.push_back(&expr);
    }

    void visit(RConstantExpr& expr, RVisitParam*) {
        refs.push_back(&expr);
    }

    // visit functions for other constructs and expressions
    ...
}
```

Condition Insertion Visitor

To change an AndOrTable truth value of `*` to `T` or `F`, the Visitor needs to keep track of all the pointers to the AndOrTables in a RSML^{-e} specification. We use a vector `allTables` for this purpose. A mutation target is selected by randomly choosing a table from this vector and then randomly choosing an index for the truth values. If the selected truth value is `*`, the mutation is performed. However, if the truth value is not `*`, this selection is discarded

and another random selection is made until a * is found. The C++ style pseudocode for the `RCondInsertVisitor` is shown below:

```
#include <vector>
#include <cstdlib>

class RCondInsertVisitor {
private:
    RComponent *fComponent;
    vector<RAndOrTable*> allTables;
    unsigned int faultNumber;

public:
    RCondInsertVisitor(unsigned int number) {
        fComponent = NULL;
        faultNumber = number;
    }

    virtual ~RCondInsertVisitor() { }

    void execute(RComponent* component) {
        RCloneComponentPass1Visitor p1_visitor;
        RCloneComponentPass2Visitor p2_visitor;
        RComponent *fComponentCopy;
        RTwoComponentMap* componentMap;
        unsigned int random1, random2, random3, random4, flag = 0;

        fComponent = component;

        while (i < faultNumber) {
            // copy the AST
            fComponentCopy = new RComponent();
            componentMap = new RTwoComponentMap(fComponentCopy);
            p1_visitor.execute(fComponent, newComponent, componentMap);
            p2_visitor.execute(fComponent, newComponent, componentMap);

            // rename the AST copy
            stringstream ss;
            ss << fComponent.name() << "_CondInsertFault" << i;
            fComponentCopy->setName(ss.str());

            // attempt to visit state variable definitions
            {
                StateVariableSet::iterator it;
                for (it = fComponentCopy->stateVariables().begin();
                    it != fComponentCopy->stateVariables().end();
                    it++) {
                        attemptVisit(*it);
                    }
            }

            // attempt to visit receive type input interfaces
            {
                InputReceiverSet::iterator it;
                for (it = fComponentCopy->inputReceivers().begin();
                    it != fComponentCopy->inputReceivers().end();
                    it++) {
```

```
                attemptVisit(*it);
            }
        }

        // attempt to visit read type input interfaces
        ...

        // attempt to visit output interfaces
        ...

        // attempt to visit macros definitions
        ...

        // attempt to visit function definitions
        ...

        // select a random cell from the tables
        random1 = rand() * allTables.size() / (RAND_MAX + 1.0);
        RAndOrTable* temp = allTables.at(random1);
        random2 = rand() * temp->height() / (RAND_MAX + 1.0);
        random3 = rand() * temp->width() / (RAND_MAX + 1.0);

        // change it to T or F if originally *
        if (temp->cell(random3, random2) == kDontCare) {
            random4 = rand() * 2 / (RAND_MAX + 1.0);
            temp->setCell(random3, random2, random4);
            flag = 1;
        }

        if (flag) {
            // typecheck the mutated AST
            RParseMessageList l;
            RPP_TypeCheck typeChecker(*newComponent, l);
            typeChecker.execute();

            if (l.errorCount == 0) {
                // print AST to RSML spec
                RPrintASTVisitor pv;
                pv.execute(fComponentCopy);
                i ++;
                flag = 0;
            }
        }

        delete fComponentCopy;
        delete componentMap;
    }
}

void visit(RStateVariable& sv, RVisitParam*) {
    // iterate through the cases and attempt to visit
    // the associated AndOrTables
}

void visit(RAndOrTable& table, RVisitParam*) {
    allTables.push_back(&table);
}
```

```
    // visit functions for other constructs
    ...
}
```

Condition Removal Visitor

The tasks performed to create a condition removal mutation in an AndOrTable are similar to that of the Condition Insertion Visitor. So it can be grouped with the previous example, with an added class variable indicating the type of operation to be performed.

Condition Negation Visitor

Similarly, the Condition Negation Visitor can be grouped with the previous two visitors.

Literal Replacement Visitor

To mutate a literal value into another value of a compatible type, the Literal Replacement Visitor gathers all the literal expression occurrences into a vector `allLiterals`, randomly select an element from this vector and randomly mutate it to a value of the same type. `Int` and `Real` type literals will not be mutated. The C++ pseudocode for the `LitReplaceVisitor` class is shown below:

```
class RLitReplaceVisitor {
private:
    RComponent *fComponent;
    vector<RLiteralValueExpr*> allLiterals;
    unsigned int faultNumber;

public:
    RLitReplaceVisitor(unsigned int number) {
        fComponent = NULL;
        faultNumber = number;
    }

    virtual ~RLitReplaceVisitor() { }

    void execute(RComponent* component) {
        RCloneComponentPass1Visitor p1_visitor;
        RCloneComponentPass2Visitor p2_visitor;
        RComponent *fComponentCopy;
        RTwoComponentMap* componentMap;
        unsigned int random1, random2, flag = 0;
```

```
fComponent = component;

while (i < faultNumber) {
    // copy the AST
    fComponentCopy = new RComponent();
    componentMap = new RTwoComponentMap(fComponentCopy);
    p1_visitor.execute(fComponent, newComponent, componentMap);
    p2_visitor.execute(fComponent, newComponent, componentMap);

    // rename the AST copy
    stringstream ss;
    ss << fComponent.name() << "_LitReplaceFault" << i;
    fComponentCopy->setName(ss.str());

    // attempt to visit state variable definitions
    {
        StateVariableSet::iterator it;
        for (it = fComponentCopy->stateVariables().begin();
            it != fComponentCopy->stateVariables().end();
            it++) {
                attemptVisit(*it);
            }
    }

    // attempt to visit input variable definitions
    {
        InputVariableSet::iterator it;
        for (it = fComponentCopy->inputVariables().begin();
            it != fComponentCopy->inputVariables().end();
            it++) {
                attemptVisit(*it);
            }
    }

    // attempt to visit receive type input interfaces
    {
        InputReceiverSet::iterator it;
        for (it = fComponentCopy->inputReceivers().begin();
            it != fComponentCopy->inputReceivers().end();
            it++) {
                attemptVisit(*it);
        }
    }

    // attempt to visit read type input interfaces
    ...

    // attempt to visit output interfaces
    ...

    // attempt to visit macros definitions
    ...

    // attempt to visit function definitions
    ...

    // attempt to visit constant definitions
```

```
...

random1 = rand() * allLiterals.size() / (RAND_MAX + 1.0);
RLiteralValueExpr* temp = allLiterals.at(random1);

// mutate the literal value for enum and bool types
if (temp->value().type()->type() == RType::kEnum) {
    if (!temp->value().undefined()) {
        random2 = rand() * (((RTypeEnum*)temp->value().type())->elements().size()
                + 1.0) / (RAND_MAX + 1.0);

        if (random2 == ((RTypeEnum*)temp->value().type())->elements().size()) {
            temp->value().makeUndefined();
            flag = 1;
        }
        else if (value.enumVal() != random2) {
            temp->value().setEnumVal(random2);
            flag = 1;
        }
    }
    else {
        random2 = rand() * ((RTypeEnum*)temp->value().type())->elements().size()
                 / (RAND_MAX + 1.0);

        // makeDefined() function not available now, should be added to RValue
        temp->value.makeDefined();
        temp->value.setEnumVal(random2);
        flag = 1;
    }
}
else if (temp->value().type()->type() == RType::kBool) {
    if (!temp->value().undefined()) {
        random2 = rand() * 2 / (RAND_MAX + 1.0);

        if (random2 == 1) temp->value().makeUndefined();
        else temp->value().setBoolVal(!temp->value().boolVal());
        flag = 1;
    }
    else {
        random2 = rand() * 2 / (RAND_MAX + 1.0);

        temp->value.makeDefined();
        if (random2) temp->value().setBoolVal(true);
        else temp->value().setBoolVal(false);
        flag = 1;
    }
}

if (flag) {
    // print AST to RSML spec
    RPrintASTVisitor pv;
    pv.execute(fComponentCopy);
    i ++;
    flag = 0;
}

delete fComponentCopy;
```

```
            delete componentMap;
        }
    }

    void visit(RStateVariable& sv, RVisitParam*) {
        attemptVisit(sv.initialValue());

        vector<RCase*>::iterator iter;
        for (iter = sv.cases().begin(); iter != sv.cases().end(); iter ++) {
            attemptVisit((*iter)->assignExpression());
            attemptVisit((*iter)->condition()->andOrTable());
        }
    }

    void visit(RAndOrTable& table, RVisitParam*) {
        for (int i = 0; i < table.height(); i ++)
            attemptVisit(table.predicate(i));
    }

    void visit(RLiteralValueExpr& expr, RVisitParam*) {
        allLiterals.push_back(&expr);
    }

    // visit functions for other constructs and expressions
    ...
}
```

Stuck-At Visitor

The Struck-at Visitor gathers all the parent nodes of the `RCondition` objects (including `RCase`, `RMacro`, and `RIOActionHandler` objects), randomly select one and replace its child condition node with a new `RCondition` object that represents `True` or `False`. The C++ style pseudocode for `StuckAtVisitor` is shown below:

```
class RStuckAtVisitor { private:
    RComponent *fComponent;
    vector<RDataStructureObject*> allConditions;
    unsigned int faultNumber;

public:
    RStuckAtVisitor(unsigned int number) {
        fComponent = NULL;
        faultNumber = number;
    }

    virtual ~RStuckAtVisitor() { }

    void execute(RComponent* component) {
        RCloneComponentPass1Visitor p1_visitor;
        RCloneComponentPass2Visitor p2_visitor;
        RComponent *fComponentCopy;
        RTwoComponentMap* componentMap;
```

```
unsigned int random1, random2, flag = 0;

fComponent = component;

while (i < faultNumber) {
    // copy the AST
    fComponentCopy = new RComponent();
    componentMap = new RTwoComponentMap(fComponentCopy);
    p1_visitor.execute(fComponent, newComponent, componentMap);
    p2_visitor.execute(fComponent, newComponent, componentMap);

    // rename the AST copy
    stringstream ss;
    ss << fComponent.name() << "_LitReplaceFault" << i;
    fComponentCopy->setName(ss.str());

    // attempt to visit state variable definitions
    {
        StateVariableSet::iterator it;
        for (it = fComponentCopy->stateVariables().begin();
            it != fComponentCopy->stateVariables().end();
            it++) {
                attemptVisit(*it);
            }
    }

    // attempt to visit receive type input interfaces
    {
        InputReceiverSet::iterator it;
        for (it = fComponentCopy->inputReceivers().begin();
            it != fComponentCopy->inputReceivers().end();
            it++) {
                attemptVisit(*it);
        }
    }

    // attempt to visit read type input interfaces
    ...

    // attempt to visit output interfaces
    ...

    // attempt to visit macros definitions
    ...

    // attempt to visit function definitions
    ...

    random1 = rand() * allLiterals.size() / (RAND_MAX + 1.0);
    RMacro* targetMacro = dynamic_cast<RMacro*>(allConditions.at(random1));
    RCase* targetCase = dynamic_cast<RCase*>(allConditions.at(random1));
    RIOActionHandler* targetHandler = dynamic_cast<RIOActionHandler*>(allConditions.at(random1));

    RliteralValueExpr* literal;
    random2 = rand() * 2 / (RAND_MAX + 1.0);
    if (random2) {
        RValue trueVal(true);
```

```
            literal = new RLiteralValueExpr(trueVal);
        }
        else {
            RValue falseVal(false);
            literal = new RliteralValueExpr(falseVal);
        }

        if (targetMacro) {
            delete targetMacro->condition();
            targetMacro.setCondition(new RCondition(literal));
            flag = 1;
        }
        else if (targetCase) {
            delete targetCase->condition();
            targetCase->setCondition(new RCondition(literal));
            flag = 1;
        }
        else if (targetHandler) {
            delete targetHandler->condition();
            targetHandler->setCondition(new RCondition(literal));
            flag = 1;
        }

        if (flag) {
            // typecheck the mutated AST
            RParseMessageList l;
            RPP_TypeCheck typeChecker(*newComponent, l);
            typeChecker.execute();

            if (l.errorCount == 0) {
                // print AST to RSML spec
                RPrintASTVisitor pv;
                pv.execute(fComponentCopy);
                i ++;
                flag = 0;
            }
        }

        delete fComponentCopy;
        delete componentMap;
    }
}

void visit(RStateVariable& sv, RVisitParam*) {
    vector<RCase*>::iterator iter;

    for (iter = sv.cases().begin(); iter != sv.cases.end(); iter ++)
        allConditions.push_back(*iter);
}

void visit(RMacro& sv, RVisitParam*) {
    allConditions.push_back(&sv);
}

void visit(RInputReceiver& ir, RVisitParam*) {
    vector<RIOActionHandler*>::iterator iter;
```

```
        for (iter = ir.receiveHandlers().begin();
             iter != ir.receiveHandlers().end(); iter ++)
             allConditions.push_back(*iter);
    }

    // visit functions for other constructs
    ...
}
```

4.3 Current Status

Up to now we have implemented the four fault-seeding visitors that relate to fault types we observed during the original development: the Variable Replacement Faults, the Condition Insertion Faults, the Condition Removal Faults, and the Condition Negation Faults. One hundred faults of the four types in RSML^{-e} specifications were automatically generated and used in Lurch random search experiments.

Bibliography

[1] D. R. Kuhn. *Fault classes and error detection capability of specification-based testing.* ACM Transactions on Software Engineering Methodology, 8(4), pages 411-424, October 1999.

[2] T. Tsuchiya and T. Kikuno. *On fault classes and error detection capability of specification-based testing.* ACM Transactions on Software Engineering Methodology, 11(1), pages 58-62, January 2002.

[3] P. E. Ammann, P. E. Black, and W. Majurski. *Using model checking to generate tests from specifications.* In Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98), pages 46-54. IEEE Computer Society, Dec. 1998.

[4] P. E. Black, V. Okun, and Y. Yesha. *Mutation operators for specifications.* Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on , 2000 Pages 81-88.

[5] P. E. Ammann and P. E. Black. *A specification-based coverage metric to evaluate test sets.* In Proceedings of Fourth IEEE International High-Assurance Systems Engineering Symposium (HASE 99), pages 239-248. IEEE Computer Society, November 1999.

[6] K. Tewary and M. J. Harrold. *Fault modeling using the program dependence graph.* Proceedings of the Fifth International Symposium on Software Reliability Engineering, November 1994, pp. 126-135.

[7] F. D. Frate, *et al. On the correlation between code coverage and software reliability.* Proceedings of the Sixth International Symposium on Software Reliability Engineering, October 1995, pp. 124-132.

[8] A. Pasquini, E. D. Agostino, and G. Di Marco. *An input-domain based method to estimate software reliability.* IEEE Transactions on Reliability, March 1996, pp. 95-105.